Implementing Fractals with MMX™ Technology

Information for Developers and ISVs

From Intel® Developer Services
www.intel.com/IDS

March 1996

CONTENTS

1.0 Introduction

The Intel Architecture MMX™ technology extensions use a Single Instruction Multiple Data (SIMD) technique to increase the efficiency of the software by operating on multiple data elements in parallel. In addition to the scalar registers the availability of eight 64-bit registers can potentially decrease the usage of memory as temporary space thus reducing costly memory accesses. This application note illustrates how to use the MMX technology to achieve better performance in generating Mandelbrot Set fractals. In graphics applications fractal geometry methods are used to realistically describe and generate natural objects such as mountains and clouds. Unlike continuous Euclidean shapes fractal objects retain their sharpness on magnification, such that as the camera gets closer to an object smaller detail becomes more visible. In this paper the code for an MMX technology function, a scalar assembly function, and a floating point function are presented, and performance and accuracy trade-offs are explained. Note that the concepts and coding tricks used in this paper are applicable to fractal functions other than the Mandelbrot Set. Figure 1 shows a sample Mandelbrot Set fractal.

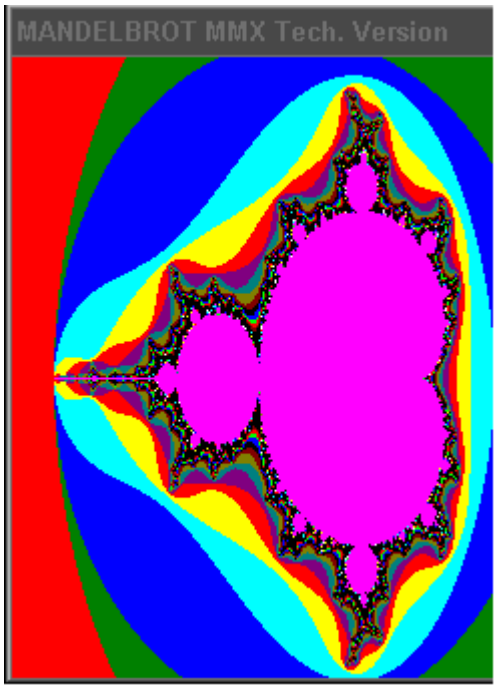

Figure 1: Sample Mandelbrot Set Fractal

2.0 The Mandelbrot Set

Fractal geometry uses procedures to describe fragmented or irregular shaped objects. In theory these objects are represented with procedures which are repeated infinitely. However, for graphical applications, the procedures are run for a finite count. There are several classes of fractals, and one of these uses nonlinear transformations named "invariant fractal sets". The Mandelbrot set, which is a self-squaring fractal, is included in this class.

The Mandelbrot set is the set of complex values that do not diverge under the squaring transformation. In this application note, a complex number, z is defined to be an ordered pair of real numbers and is defined as:

$$z=x + jy \text{ ;where } x=Re(z),\ y=Im(z)$$

The real and imaginary parts will further be referenced as *c.r* and *c.i* respectively.

Thus the squaring transformation for the Mandelbrot set can be written as:

$z_0 = z$
$z_k = z^2_{k-1} + z_0$, k=1,2,3,...

z_k is repeatedly calculated until it can be determined whether or not the transform is diverging.The fractal is the boundary of the convergence region in the complex plane.

To generate the Mandelbrot set a window is chosen in the complex plane. The major part of the set is in the region:

-2.25 <= Re(z) <= 0.75
-1.25 <= Im(z) <= 1.25

By selecting smaller windows in this range one can effectively zoom in and generate more detailed images. The algorithm maps the positions in this region to color coded pixel positions on the display surface. By using the magnitude of the complex number one can determine whether or not the number will diverge quickly. One can set a magnitude limit and iterate until this limit is reached or an arbitrary iteration limit is reached. The iteration count then can be used as an index to a color palette and the resulting pixel color is put on the display surface. The high level algorithm implemented in C is given below:

```
void mandelbrot ( rMin, rInc, iMin,  iInc,  cols,  rows)
{

        int xc, yc, ccount;
        COMPLEX zInit;
        PALETTE  LocalPalette[256];
//      rInc is defined as  (rMax - rMin) /cols;
//      iInc is defined as  (iMax - iMin) /rows;
        zInit.r = rMin;
        for (xc=0; xc<cols; xc++)
        {
                zInit.i  = iMin;
                for (yc=0; yc<rows; yc++)
```

```
                {
                        ccount = Iterate(zInit);
                        PutPixel (xc, yc, LocalPalette[ccount]);
                        zInit.i = zInit.i + iInc;
                }
                zInit.r = zInit.r + rInc;
        }

}
int Iterate (COMPLEX zInit)
{
        int cnt= 0;
        COMPLEX z;
        z= zInit;
        do
        {
                z = ComplexSqr(z);
                c.r = c.r + zInit.r;
                c.i = c.i + zInit.i;
                cnt++;
        }
        while ((c.r*c.r+c.i*c.i <=4.0) & (cnt <255));
        return cnt;
}
COMPLEX ComplexSqr(COMPLEX c)
{
        COMPLEX result;
        result.x =  c.r * c.r - c.i * c.i ;
        result.y =  2 *  c.r * c.i ;
        return result;
}
```

Figure 2. The High Level Algorithm Implemented in C

The Mandelbrot function as shown in Figure 2 calculates the color intensity for each pixel and maps them to the display surface. The size of the display surface is defined by the row and column size, and the complex window is defined by the complex values passed in to the function. The color intensity value is returned by the iterate function. The maximum limit for the iteration count is set to 256, which will give 256 different colors with a 256 color palette. At each iteration the complex number is squared and incremented by the delta values. Then the magnitude for the resulting complex number is calculated and checked against the value 4.0. If the magnitude is equal or higher than 4.0 the iteration count is returned. In the event that the magnitude does not diverge the maximum iteration count is returned.

3.0 Input and Output Data Representation

The real and imaginary parts of the complex numbers are each represented as a word (16-bits) in fixed point notation 5.11 (five bits for the whole part and eleven bits for the fractional part). This format was chosen because the window for the Mandelbrot set can be represented with one decimal digit for the whole part. Five bits are sufficient to represent the whole part of the signed number in binary. Eleven fractional bits provide a reasonably sharp image. As sections of the image are enlarged a certain loss of precision is possible.

By using words to represent the real and imaginary parts, two complex numbers can be stored in one 64-bit register, as shown in Figure 3. It is desirable to store multiple numbers in the registers so that they can be manipulated in parallel with MMX instructions.

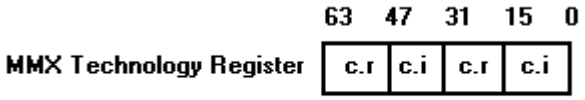

Figure 3: Complex Numbers in 64-bit Register

The loop iteration count, which also represents the color intensity index, is an unsigned byte. Even though eight bits are enough to access the palette a 32-bit integer register is used to avoid prefix penalties. The color palette is an array of 256 24-bit RGB values derived from the Microsoft Windows System Palette.

Several MMX technology registers are used to hold intermediate values to avoid partial read/writes to memory. As discussed in Section 2.0, the algorithm requires real and imaginary increment values to be added to complex numbers in the two loops which process x- and y-coordinates. Two 64-bit registers are initialized with the appropriate values to perform the increments in one cycle. The register contents are displayed in Figure 4, below.

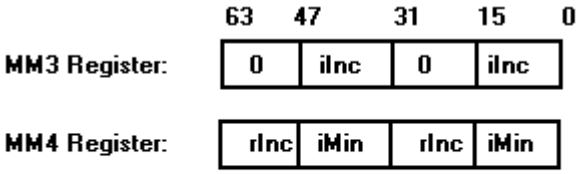

Figure 4: Increment Value Registers

Both the complex number magnitude and the iteration count need to be checked at the innermost loop. To avoid two separate compares these two values are stored in one 64-bit register, and one compare instruction is used. The register is shown in Figure 5.

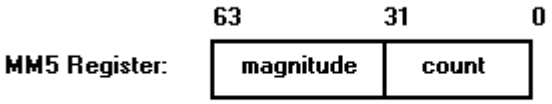

Figure 5: Magnitude/Iteration Count Register

4.0 Code Partitioning

4.1 Data Setup

In this section the data setup is explained in detail to show how to effectively pass 32-bit and 16-bit parameters to functions. The parameters to the function are:

```
mandmmx(  hdc, ptrmparams, CWIN_HEIGHT, CWIN_WIDTH, ptrPal);
```

The *hdc* is a window handle which is a 32-bit value and *ptrmparams* is a 32-bit pointer to the *MMXPARAMS* struct which contains the values for the complex window range. *CWIN_HEIGHT* and *CWIN_WIDTH* are 32-bit integer values for the display surface size, and the *ptrPal* is a 32-bit pointer to the palette used. Please note that all parameters passed to the function are 32-bit aligned. The structure, *MMXPARAMS*, is of particular interest since by setting the data up properly we can avoid partial memory accesses and also read data 64-bits at a time. The structure is 16-bit word aligned. As explained in Section 3 the Mandelbrot MMX technology code operates on 16-bit words, however, passing in the parameters as 16-bit values would have been costly due to partial reads. The structure is defined as:

```
typedef struct defMMXPARAMS
{
        int mm2_1;      //rMin
        int mm2_2;      //iMin
        int mm3_1;      //0
        int mm3_2;      //iInc
        int mm4_1;      //rInc
        int mm4_2;      //iMin
} MMXPARAMS;
```

As the structure element names imply, the first two values are loaded as a 64-bit value into the register mm2, the next two into mm3, and the last two into mm4. A *packssdw* instruction is performed on all three registers to set the data up as 16-bit words. The data setup code is given in Figure 6. Memory is accessed sequentially to shorten memory access time. Instructions are out of order for proper pairing.

```
 mov edi, mparams       ;pointer to mparams
 xor edx, edx           ;clear x-coor counter
 movq mm2, [edi]        ;load mm2       rMin iMin

 movq mm3, [edi+8]      ;load mm3       0    iInc
 packssdw mm2, mm2      ;mm2 =  rMin iMin rMin iMin
 movq mm4, [edi+16]     ;load mm4       rInc iMin
 packssdw mm3, mm3      ;mm3 =  0    iInc 0    iInc

 packssdw mm4, mm4      ;mm4 =  rInc iMin rInc iMin
```

Figure 6: Data Setup Code

The registers are formatted as shown below to facilitate efficiency in the code. The reason this format was chosen will be evident when the code is discussed later on.

```
        after loading          after packing to 16-bits
MM2     rMin iMin               rMin iMin rMin iMin
MM3     0    iInc               0    iInc 0    iInc
MM4     rInc iMin               rInc iMin rInc iMin
```

4.2 Inner Loop Section

The MMX technology code is written using the algorithm given in Figure 2. At the top level there are two loops, one incrementing the x-coordinate and the other incrementing the y-coordinate. The iteration loop where the code spends most of its time is the innermost loop. A section of code is used to set up the stack for the Windows function to display the pixel on the surface. Each of these sections will be described below.

4.2.1 Top Level Loops

The code section in Figure 7 shows the two top level loops. The code's efficiency is contained in the method that the increments are performed. The code shown is not optimized.

```
loopx:
        movq mm0, mm2           ;mm0 gets a copy of m2
        xor ecx, ecx            ;clear y coor counter
        loopy:
                pxor mm5, mm5   ;clear count and magnitude register
                xor eax, eax    ;color/count register

                iter:
                        ;;This is the iteration loop
                doneiter:
                        ;Set pixel data and make the call
                        call    SetPixelV@16    ;call to SetPixelV
                paddsw mm2, mm3         ;add iInc to mm2
                movq mm0, mm2           ;mm0 gets m2

                inc ecx                 ;increment yloop counter
                cmp ecx, rows           ;done yet?
                jne loopy
        pand mm2, dword ptr EVENWORDMASK        ;mm2 rval 0 rval 0
        paddsw mm2, mm4         ;add rInc to mm2
        inc edx                 ;increment x loop counter
        cmp edx, cols           ;done yet?
        jne loopx
```

Figure 7: The Top Level Loops

4.2.2 Iteration Loop

In the iteration loop, MMX instructions are used to operate on multiple data in parallel to speed up the calculations. For instructional purposes the code given here is not optimized. The performance optimized code, which is given in Appendix A, needs to be examined carefully to understand the high instruction per clock ratio of this particular section.

```
iter:
        movq mm1, mm0           ;mm0=mm1 is c.r c.i c.r c.i
        pmullw mm1, dword ptr NEGMASK ;  mm1 = c.r -c.i c.r c.i
        pmaddwd mm1, mm0        ;multiply add mm1, mm0
        psrad mm1, 11           ;adjust precision for fix pt
        punpckldq mm5, mm1      ;mm5 was 0 curcnt
                                ;mm5 is now check curcnt
        pcmpgtd mm5, dword ptr CHECKCTRMASK
                                ; check if mm5 greater than 4.0 255
                                ; if mm5 has any 1's then
                                ; either condition was true
        packssdw mm5, mm5       ;pack upper and lower dwords into words
```

```
movd ebx, mm5          ;ebx gets mm5
cmp ebx, 0h            ;check if ebx is all zeros
jne doneiter           ;if not we are done iterating

inc eax                ;increment color/counter register
movd mm5, eax          ;lower dword of mm5 is color/cnter value
psrlq mm1, 32          ;mm1 0 res.r
movq mm6,mm0           ;mm6 gets a copy of mm0
psrlq mm0, 16          ;mm0 = 0 c.r c.i c.r
pand mm0, dword ptr ODDWORDMASK ;mm0 = 0 c.r 0   c.r
pmaddwd mm0, mm6       ;mm0 is 0+c.r*c.i 0+c.r*c.i
psrad  mm0, 10         ;adjust precision for fx pt
                       ;actually (fracbits-1) for the multiply.
punpckldq mm0, mm1     ;mm0 res.r res.i
packssdw mm0, mm0      ;mm0 is res.r res.i res.r res.i
paddsw  mm0,mm2        ;mm1 is z^2+z0
jmp iter
```

Figure 8: The Iteration Loop

There is no pairing shown in the above code. Figure 8 is a translation of the code in Figure 2 to MMX technology without regard to optimization. Note that in the first section with one *pmaddwd* instruction both the *result.r* and the *magnitude* are calculated. Also, by using the mm5 register to perform the compare both the magnitude and the maximum iteration count are checked in parallel. During the calculation of *result.i* the multiply by two is done without an extra shift or multiply instruction by shifting right one less than the number of fractional bits.

Even in this form, the MMX technology code will be quite efficient However, there are register dependencies on registers mm0, mm1, and mm5 since the operations are performed sequentially. By taking some of the instructions which work on mm0 and interleaving them with the instructions which use mm1 and mm5, the pairing can be increased while lowering penalties. The resulting code with the optimal pairing is in Appendix A.

4.2.3 Pixel Display

Pixel mapping to the display surface is accomplished by using the Windows API function SetPixelV. The parameters for this function must be pushed on the stack before the call is made. The integer registers are not preserved on returning from the function. Note that using this function is not the most optimal way of putting the pixels on the display surface. However, it keeps the overall algorithm simpler. It is also possible to store the pixel information in the memory and then use a high level block transfer function to update the display. Interested readers are encouraged to try this approach.

The sequence to call the SetPixelV is given below:

```
mov eax, PPal[eax]     ;index to the palette
mov     ebx,hdc        ;the handle
push    eax            ;the color
push    ecx            ;the y coor
push    edx            ;the x coor
push    ebx
call    SetPixelV@16   ;call to SetPixelV
                       ;@16 is reqired when the function
                       ;is called from an assembly routine.
```

Figure 9: Pixel Display

4.3 Code Comparison

The Mandelbrot algorithm shown in Figure 2 has also been implemented using integer assembly and floating point code. These two versions are also included in Appendix A.

Note that the integer scalar code uses 32-bit registers but the fixed point format is left as 5.11. Prefix penalties are avoided by using 32-bit registers and 32-bit instructions. Since there are different exit conditions from the innermost loop, it is very difficult to save variables on the stack and retrieve them. Thus, memory is used to store most of the intermediate values. Wherever possible registers are saved on the stack. There is no apparent way to operate on two complex numbers simultaneously. However, with careful use of all available general purpose registers, penalties are avoided. Also note that *imul* instructions take ten clocks, as opposed to the single clock needed to issue *pmaddwd* instructions.

The floating point code uses single precision numbers. The code flow is similar to the integer version. One of the drawbacks is having to perform operations on the top of the stack as mandated by the definition of the instructions. This increases dependency on previous values and causes pipeline stalls. Additionally, saving from the top of the stack to memory takes several clock cycles. Floating point multiply instructions take three clocks each as opposed to the ten required by *imul*.

4.4 Sample Code

The executable file for the Mandelbrot Set is also included for downloading. This code is meant to be run under Microsoft Windows '95 OS on an Intel system with an MMX technology processor, and has not been tested on any other systems. The program can be run to obtain the number of CPU clocks execution takes with different complex windows.

The user can select either default parameters or custom parameters for the complex window. Once parameters are entered, the user executes the "run" instruction and the display is updated.

5.0 Performance

All three versions of the Mandelbrot function are optimized for speed. Memory accesses are 32-bit aligned and the instructions are paired where possible. The timings were measured with Intel's VTune 2.0 [please see the VTune Home Page for information on ordering this product.]. The application also includes a utility which measures the number of CPU ticks taken by the different functions. The static analysis data including timing and pairing information obtained with VTune is for the functions presented in Appendix A. The clock cycles reported by the shell program include all the clock cycles for data setup and call for the individual functions in the C portion. They also show all the clocks needed to draw the complete object on the display surface using the SetPixelV function, thus they reflect real execution time. All performance numbers are based on a prototype 150MHz Pentium(R) Processor with MMX technology.

Using the default parameters for the complex window given in Section 2, and using 320 for CWIN_WIDTH and 240 for CWIN_HEIGHT, the integer assembly version takes about 425 million clocks, the floating point version takes about 407 million clocks, and the MMX technology version uses about 286 million clocks. By changing the size of the complex window different clock counts are obtained. With default parameters the MMX technology code is 42 % faster than the floating point code and 49 % faster than the integer code. With certain complex windows the MMX technology version will perform upto 100% faster than the other two versions. The clocks per pixel is high due to the inefficiency of the SetPixelV function. However, the SetPixelV function adds the same number of cycles to each function.

The values obtained using VTune are given in the tables below. The first table contains the three functions outlined in the text. The second table lists the C-functions which are used to arrange the parameters and call the functions listed in the first table. Note that the calling functions for integer and MMX technology versions use more instructions for data setup for the fixed point conversion. Effects of cache misses and other dynamic issues are not discussed since they do not play a major role with the algorithm used. The color palette has only 256 entries and is probably in the cache most of the time.

Function name	Description	Instr	Pairing	Clocks
mandasm	Integer assembly	75	75%	85
mandfpu	Floating pt.	79	48%	101
mandmmx	MMX technology	68	76%	51

Function Description	Clocks per Pixel
C calling function to measure integer Mandelbrot Set function	5533
C calling function to measure fpu Mandelbrot Set function	5299
C calling function to measure MMX tech. Mandelbrot Set function	3724

Table 1-2) Performance Numbers

APPENDIX A

```
TITLE   mandmmx
;*************************************************************************/
;*
;*      This program has been developed by Intel Corporation.
;*       You have Intel's permission to incorporate this code
;*      into your product, royalty free.  Intel has various
;*       intellectual property rights which it may assert under
;*      certain circumstances, such as if another manufacturer's
;*      processor mis-identifies itself as being "GenuineIntel"
;*       when the CPUID instruction is executed.
;*
;*      Intel specifically disclaims all warranties, express or
;*      implied, and all liability, including consequential and
;*       other indirect damages, for the use of this code,
;*       including liability for infringement of any proprietary
;*       rights, and including the warranties of merchantability
;*       and fitness for a particular purpose.  Intel does not
;*       assume any responsibility for any errors which may
;*       appear in this code nor any responsibility to update it.
;*
;*  *  Other brands and names are the property of their respective
;*     owners.
;*
;*************************************************************************/
;  This program was assembled with Microsoft MASM 6.11d
;
; prevent listing of iammx.inc file
.nolist
 INCLUDE iammx.inc                  ; IAMMX   Macros
.list
.586
.model FLAT, STDCALL
extern SetPixelV@16:proc          ;Needed to put pixels on the surface
;*************************************************************************
;    Data Segment Declarations
;*************************************************************************
.data
ODDWORDMASK      QWORD    0000FFFF0000FFFFh
EVENWORDMASK     QWORD    0FFFF0000FFFF0000h
EVENDWORDMASK    QWORD    0FFFFFFFF00000000h
NEGMASK          QWORD    0001FFFF00010001h
CHECKCTRMASK     QWORD    00001FFF000000FFh
FOUR             DD       4.0; do not delete the period.
;Local vars for the mandfpu
fzinitX          DD               ?
fzinitY          DD               ?
fzX              DD               ?
fzY              DD               ?
frInc            DD               ?
fiInc            DD               ?
;Local vars for the mandasm
zinitX           DWORD            ?
zinitY           DWORD            ?
zX               DWORD            ?
zY               DWORD            ?
temp             DWORD            ?
temp1            DWORD            ?
;*************************************************************************
;    Constant Segment Declarations
;*************************************************************************
.const
```

```
;*************************************************************************
;    Code Segment Declarations
;*************************************************************************
.code
COMMENT ^
void mandmmx (
        int   *hdc,
        int   mparams
        int   cols,
        int   rows,
        int   PPal) ;
^
mandmmx PROC NEAR C USES  eax ebx ecx edx,
                hdc: PTR DWORD,
                mparams:  DWORD,
                cols: DWORD, rows: DWORD, PPal: DWORD

 mov edi, mparams        ;pointer to mparams
 xor edx, edx            ;clear x-coor counter
 movq mm2, [edi] ;load mm2        rMin iMin

 movq mm3, [edi+8]       ;load mm3        0    iInc
 packssdw mm2, mm2       ;mm2 =   rMin iMin rMin iMin
 movq mm4, [edi+16]      ;load mm4        rInc iMin
 packssdw mm3, mm3       ;mm3 =   0    iInc 0    iInc

 packssdw mm4, mm4       ;mm4 =   rInc iMin rInc iMin
loopx:
        movq mm0, mm2            ;mm0 gets a copy of m2
        xor ecx, ecx             ;clear y coor counter
        loopy:
                pxor mm5, mm5    ;clear count and magnitude register
                xor eax, eax     ;color/count register

                iter:
                        movq mm1, mm0            ;mm0=mm1 is c.x c.y c.x c.y
                        pmullw mm1, dword ptr NEGMASK ;  mm1 = c.x -c.y c.x c.y
                        movq mm6,mm0             ;mm6 gets a copy of mm0
                        pmaddwd mm1, mm0 ;multiply add mm1, mm0
                        psrlq mm0, 16            ;mm0 = 0 c.x c.y c.x
                        pand mm0, dword ptr ODDWORDMASK ;mm0 = 0 c.x 0   c.x
                        psrad mm1, 11            ;adjust precision for fix pt
                        punpckldq mm5, mm1       ;mm5 was 0 curcnt
                                                 ;mm5 is now check curcnt
                        pmaddwd mm0, mm6 ;mm0 is 0+c.x*c.y 0+c.x*c.y
                        pcmpgtd mm5, dword ptr CHECKCTRMASK
                                                 ; check if mm5 greater than 4.0 255
                                                 ; if mm5 has any 1's then
                                                 ; either condition was true
                        psrlq mm1, 32            ; mm1 0 res.x
                        packssdw mm5, mm5        ;pack upper and lower dwords into words
                        psrad  mm0, 10           ;adjust precision for fx pt
                                                 ;actually (fracbits-1) for the multiply.
                        movd ebx, mm5            ;ebx gets mm5
                        cmp ebx, 0h              ;check if ebx is all zeros
                        jne doneiter             ;if not we are done iterating

                        inc eax                  ;increment color/counter register
                        punpckldq mm0, mm1       ;mm0 res.x res.y
                        movd mm5, eax            ;lower dword of mm5 is color/cnter value
                        packssdw mm0, mm0        ;mm0 is res.x res.y res.x res.y
                        paddsw   mm0,mm2         ;mm1 is z^2+z0
                        jmp iter
                doneiter:
```

```
                        ;Set pixel

                        mov eax, PPal[eax]       ;index to the palette
                        push edx         ;save xcoor counter
                        push ecx         ;save the ycoor counter
                        mov     ebx,hdc          ;the handle
                        push    eax              ;the color
                        push    ecx              ;the y coor
                        push    edx              ;the x coor
                        push    ebx
                        call    SetPixelV@16     ;call to SetPixelV
                        paddsw mm2, mm3          ;add iInc to mm2
                        pop ecx
                        pop edx
                        movq mm0, mm2            ;mm0 gets m2

                        inc ecx                  ;increment yloop counter
                        cmp ecx, rows            ;done yet?
                        jne loopy
                        pand mm2, dword ptr EVENWORDMASK  ;mm2 rval 0 rval 0
                        inc edx                  ;increment x loop counter
                        cmp edx, cols            ;done yet?
                        paddsw mm2, mm4          ;add rInc to mm2
                        jne loopx

Done:
    emms
    ret
mandmmx ENDP
;;*****************************************************************************
COMMENT ^
void mandasm (
        int   *hdc,
        int   mparams
        int   cols,
        int   rows,
        int   PPal) ;
^

mandasm PROC NEAR C USES  eax ebx ecx edx,
                hdc: PTR DWORD,
                mparams:  DWORD,
                cols: DWORD, rows: DWORD, PPal: DWORD

 mov edi, mparams                        ;ptr to mparams
 mov edx, [edi]
 mov zinitX, edx
 xor edx, edx                            ;clear x-coor counter
 loopx:
        mov ecx, [edi+8]                 ;imin
        mov zinitY, ecx
        xor ecx, ecx                             ;clear y coor counter
        loopy:
                mov eax, zinitX
                mov ebx, zinitY
                mov zX, eax
                mov zY, ebx
                xor eax, eax                     ;color/count register
                iter:
                        push edx
                        mov ebx, zY              ;zY into register ebx
                        mov edx, zX              ;zX into register edx
                        imul ebx, zY             ;ebx is c.y*c.y
                        imul edx, zX             ;edx is c.x*c.x
```

```
                        sar ebx, 11             ;shift for fixed pt
                        mov temp, ebx           ;temp is c.y*c.y
                        sar edx, 11             ;shift for fixed pt
                        mov temp1, edx          ;temp1 is c.x*c.x
                        add ebx, edx            ;ebx is c.x*c.x+c.y*c.y
                        pop edx
                        cmp ebx, 01FFFh         ;check if ebx is 4 or more
                        jg  doneiter            ;if so we are done iterating
                        cmp eax, 00FFh          ;have we reached the iteration count
                        jg  doneiter
                        push edx
                        mov ebx, zY             ;zY into register ebx
                        inc eax                 ;increment counter
                        imul ebx, zX            ;ebx is c*x*c.y
                        mov edx, temp1          ;edx gets temp1=c.x*c.x
                        sar ebx, 10             ;2*c.x*c.y aligned for fixed pt.
                        add ebx, zinitY         ;ebx is 2*c.x*c.y+zinitY
                        sub edx, temp           ;edx is c.x*c.x-c.y*c.y
                        mov zY, ebx             ;zY is 2*c.x*c.y+zinitY
                        add edx, zinitX         ;edx is c.x*c.x-c.y*c.y+zinitX
                        mov zX, edx             ;zX is c.x*c.x-c.y*c.y+zinitX
                        pop edx

                        jmp iter
                doneiter:
                        ;Set pixel

                        mov eax, PPal[eax]      ;index to the palette
                        push edx
                        push ecx
                        mov     ebx,hdc         ;the handle
                        push    eax             ;the color
                        push    ecx             ;the y coor
                        push    edx             ;the x coor
                        push    ebx
                        call    SetPixelV@16    ;call to SetPixelV
                        mov edx, zinitY
                        pop ecx
                        add edx, [edi+12]       ; add iInc
                        inc ecx                 ;increment yloop counter
                        mov zinitY, edx  ;save new zinitY
                        pop edx
                        cmp ecx, rows           ;done yet?
                        jne loopy
                        mov ebx, zinitX
                        inc edx                 ;increment x loop counter
                        add ebx, [edi+4] ;add rInc
                        cmp edx, cols           ;done yet?
                        mov zinitX, ebx         ;save new zinitX
                        jne loopx

Done:
    ret
mandasm ENDP
;;*****************************************************************************
COMMENT ^
void mandfpu (
        int   *hdc,
        int   mparams
        int   cols,
        int   rows,
        int   PPal) ;
^
```

```
mandfpu PROC NEAR C USES  eax ebx ecx edx,
                hdc: PTR DWORD,
                mparams:  DWORD,
                cols: DWORD, rows: DWORD, PPal: DWORD
 ;;All floating point instructions are in capital letters.
 mov edi, mparams                        ;ptr to mparams
 FINIT                                   ;initialize the Funit
 mov edx, [edi]
 xor eax, eax                            ;clear eax
 mov fzinitX, edx                        ;rMin
 mov ebx, [edi+4]
 mov frInc, ebx                          ;rInc
 mov ecx, [edi+12]
 mov fiInc, ecx                          ;iInc
 xor edx, edx                            ;clear x-coor counter
 loopx:
        mov ecx, [edi+8]         ;iMin
        mov fzinitY, ecx         ;fzinitY set to iMin
        xor ecx, ecx                     ;clear y coor counter
        loopy:
                mov eax, fzinitX
                mov ebx, fzinitY
                mov fzX, eax             ;initialize fzX with fzinitX
                mov fzY, ebx             ;initialize fzY with fzinitY
                xor ebx, ebx             ;color intensity register
                iter:
                        FLD  fzY         ;load zY to ST
                        FMUL fzY         ;ST is c.y*c.y
                        FST  ST(1)               ;ST(1) is c.y*c.y

                        FLD  fzX         ;load zX to ST
                        FMUL fzX         ;ST is c.x*c.x

                        FST  ST(3)               ;ST(3) is zX^2
                        ;ST(3) is zX^2
                        ;ST(2) is zY^2
                        ;ST(1) is zY^2
                        ;ST is zX^2
                        FADDP ST(1), ST          ;ST is c.x*c.x+c.y*c.y
                        FCOMP FOUR               ;check if ebx is 4 or more
                        ;ST(1) is zX^2
                        ;ST  is zY^2
                        FSTSW AX         ;flags stored in AX
                        test eax, 4500h          ;mag greater then 4.
                        jz  doneiter             ;if so we are done iterating
                        test eax, 4000h          ;mag equal to 4.
                        jnz  doneiter            ;if so we are done iterating
                       cmp ebx, 00FFh            ;have we maxed out on colors?
                        jg  doneiter
                        inc ebx                  ;increment color intensity
                        ;ST(1) is zX^2
                        ;ST is zY^2
                        FSUBR  ST , ST(1)        ;ST=ST(1)-ST;

                        FLD  fzX         ;ST is fzX
                        FMUL fzY         ;ST is c.x*c.y
                        FADD ST, ST(0)           ;ST is 2*c.x*c.y
                        FADD fzinitY             ;ST is 2*c.x*c.y+fzinitY
                        FSTP fzY         ;store and pop

                        FADD fzinitX             ;ST is c.x*c.x-c.y*c.y+fzinitX
                        FSTP fzX         ;store and pop
                        ;STACK is empty
```

```
                jmp iter
            doneiter:
                ;Set pixel

                mov eax, PPal[ebx]      ;index to the palette
                push edx
                push ecx
                mov     ebx,hdc         ;the handle
                push    eax             ;the color
                push    ecx             ;the y coor
                push    edx             ;the x coor
                push    ebx
                call    SetPixelV@16    ;call to SetPixelV
                FLD fzinitY             ;ST is fzinitY
                FADD fiInc              ;ST fzinitY+fiInc
                pop ecx
                pop edx
                inc ecx                 ;increment yloop counter
                cmp ecx, rows           ;done yet?
                FSTP fzinitY            ;fzinitY=fzinitY+fiInc --pop ST
                ;STACK is empty
                jne loopy
                FLD fzinitX             ;ST is fzinitX
                inc edx                 ;increment x loop counter
                FADD frInc              ;ST is zinitX+frInc
                cmp edx, cols           ;done yet?
                FSTP fzinitX            ;fzinitX=fzinitX+frInc --pop ST
                ;STACK is empty
                jne loopx

Done:
    ret
mandfpu ENDP
END
```